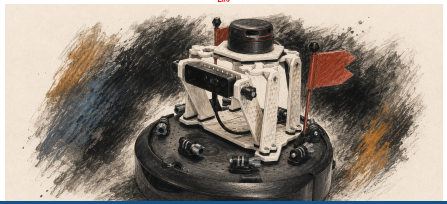
# Seguimiento de un objeto/persona con detecciones 2D/3D y evitación con láser

## Teoría y herramientas para la práctica (ROS 2 + C++)

Francisco Martín Rico y Rodrigo Pérez Rodríguez



# Qué hay que hacer en la práctica (y qué verás aquí)

- Objetivo: que el robot siga un objetivo visual y evite colisiones.
- Desarrollo incremental (valida cada paso por separado con `ros2 topic echo`, RViz o simulador):
  - (1) Láser: detectar obstáculo (`/obstacle`), calcular su (x,y) en el frame del sensor y transformarlo a `base_link` con TF.
  - (2) Cámara: detección 2D (p.ej. `/detection_2d`) con regla **si no hay detección, no publiques nada**.
  - (3) 3D: con depth + `CameraInfo`, publicar detección 3D (p.ej. `/detection_3d`) + TF propia del objetivo.
  - (4) Control: orientar hacia el objetivo (PID angular) y publicar `/cmd_vel`.
  - (5) Control de distancia: mantener 1–2 m.
  - (6) Evitación: usar `/nearest_obstacle` cuando interfiera en el seguimiento.
- Estas slides resumen la sección **Teoría para prácticas**: launchers, parámetros, `LaserScan`, TF, imágenes, depth→3D y PointCloud2→3D.

# Teoría para prácticas: mapa rápido

- **Launchers**: lanzar nodos + parámetros + remappings de forma reproducible.
- **Parámetros**: no hardcodear; YAML + `declare_parameter/get_parameter`.
- **Láser**: `LaserScan` → mínimo rango válido → (x,y) en el frame del sensor.
- **TF**: publicar frames propios y transformar puntos entre frames (con coherencia temporal).
- **Visión**: `Image` → `cv::Mat` (cv_bridge) → detecciones.
- **3D**: depth (`16UC1/32FC1`) + `CameraInfo` → (X,Y,Z); o nube organizada (`PointCloud2`).

# Launchers: lanzar nodos + parámetros + remappings

- Un **launcher** (Python) describe **cómo lanzar** un conjunto de nodos.
- Ventajas: reproducible, documenta configuración, evita comandos largos.
- Incluye: ejecutable, parámetros (YAML), remappings de topic(s), y composición (incluir otros launchers).
- Ejemplo real: `laser.launch.py` (en `ch3_examples/.../launch/`).

**Launcher mínimo (extracto)**

```
1  from launch import LaunchDescription
2  from launch_ros.actions import Node
3
4  def generate_launch_description():
5          detector_cmd = Node(
6                  package='laser',
7                  executable='
                       obstacle_detector',
8                  output='screen',
9                  parameters=[param_file],
10                 remappings=[('input_scan','
                       /scan_raw')])
11
12         ld = LaunchDescription()
13         ld.add_action(detector_cmd)
14         return ld
```

# Parámetros (YAML): configuración sin tocar código

- Los **parámetros** son configuración por nodo: umbrales, ganancias, nombres de frame, etc.
- Objetivo: **no hardcodear** constantes.
- En YAML, el nivel superior debe coincidir con el **nombre del nodo**.
- Ejemplo real (láser): `min_distance` (m) y `use_sim_time`.

**params.yaml (real)**

```
1 obstacle_detector_node:
2         ros__parameters:
3                 use_sim_time: true
4                 min_distance: 0.75
```

# Parámetros (C++): declarar + leer el valor efectivo

- Patrón recomendado:
  - `declare_parameter(name, default)`
  - `get_parameter(name, value)`
- Si el nombre del nodo en C++ no coincide con el YAML, los parámetros no se aplican.

**ObstacleDetectorNode.cpp (real)**

```
1 ObstacleDetectorNode::ObstacleDetectorNode
      ()
2 : Node("obstacle_detector_node")
3 {
4         declare_parameter("min_distance",
              min_distance_);
5         get_parameter("min_distance",
              min_distance_);
6         RCLCPP_INFO(get_logger(),
7                 "... set to %f m",
                      min_distance_);
8 }
```

# LaserScan: qué contiene y qué haces con ranges

- `LaserScan` representa un barrido láser 2D planar.
- Contiene: `ranges` (distancias), `angle_min`, `angle_increment`, y límites `range_min`/`range_max`.
- En la práctica: filtrar rangos inválidos → mínimo $r$ → $\theta = \texttt{angle\_min} + i\,\texttt{angle\_increment}$ → $x = r\cos\theta,\ y = r\sin\theta$.
- Esas coordenadas están primero en el **frame del sensor** (`header.frame_id`); luego las transformarás con TF.

# Suscripción a `LaserScan` (QoS de sensor)

- Para sensores, usa QoS de sensor (baja latencia; tolera pérdidas).
- En el ejemplo se usa `SensorDataQoS().reliable()`.
- Tu nodo del Paso 1 se suscribe al scan y calcula el obstáculo más cercano.

**Suscripción (real)**

```
laser_sub_ = create_subscription<sensor_msgs::msg::
      LaserScan>(
  "input_scan",
  rclcpp::SensorDataQoS().reliable(),
  std::bind(&ObstacleDetectorNode::laser_callback, this,
        _1));
```

# Mínimo rango → (x,y) del obstáculo más cercano

- `std::min_element` devuelve un **iterador** al mínimo.
- El **índice** se obtiene restando iteradores (o con `std::distance`).
- Luego: $\theta = \texttt{angle\_min} + i\,\texttt{angle\_increment}$ y proyección trigonométrica.
- En la práctica: filtra primero `NaN`/`Inf` y valores fuera de `range_min`/`range_max`.

**Cálculo (real)**

```
1  int min_idx = std::min_element(
2         scan.ranges.begin(),
3         scan.ranges.end()) - scan.ranges.
4              begin();
5
6  float distance_min = scan.ranges[min_idx];
7  float obstacle_angle = scan.angle_min +
8         min_idx * scan.angle_increment;
9  float x = distance_min * std::cos(
10      obstacle_angle);
   float y = distance_min * std::sin(
       obstacle_angle);
```

# `std::min_element`: alternativa legible con `std::distance`

- A veces es más claro separar pasos: primero el iterador al mínimo, luego el **índice**.
- `std::distance(begin, it)` devuelve cuántas posiciones hay desde `begin` hasta `it`.

**Alternativa (real)**

```
auto it_min = std::min_element(scan.ranges
     .begin(), scan.ranges.end());
int min_idx = std::distance(scan.ranges.
     begin(), it_min);
```

# TF (`tf2`): marcos de referencia y coherencia temporal

- TF gestiona transformaciones entre **frames**: padre → hijo.
- Puedes publicar un frame propio (p.ej. `target`) o transformar puntos a un frame común (p.ej. `base_link`).
- Idea clave de la práctica: si calculas algo con un sensor a un tiempo $t$ (p.ej. `LaserScan.header.stamp`), la TF/punto que publiques debe ser coherente con ese $t$.

# Publicar una TF (TransformBroadcaster)

- Se crea un TransformBroadcaster y se envía un TransformStamped.
- Campos clave:
  - header.frame_id: frame padre
  - child_frame_id: frame hijo
  - header.stamp: instante en el que la TF es válida
- En tu Paso 1/3, publicarás frames propios para obstáculo/objetivo.

**TFPublisherNode.cpp (real)**

```
1  tf_broadcaster_ = std::make_shared<
2         tf2_ros::TransformBroadcaster>(*
              this);
3
4  transform_.header.frame_id = "odom";
5  transform_.child_frame_id = "target";
6  transform_.transform.translation.x = ...;
7  transform_.transform.translation.y = ...;
8  transform_.transform.translation.z = 0.0;
9
10 transform_.header.stamp = now();
11 tf_broadcaster_->sendTransform(transform_)
       ;
```

# Leer una TF (Buffer + TransformListener)

- Patrón seguro:
  - `canTransform(...)` para evitar excepciones
  - `lookupTransform(...)` para obtener la transformación
- `TimePointZero` pide "la última disponible".
- Si necesitas coherencia temporal, consulta con el timestamp del sensor.

**TFSeekerNode.cpp (real)**

```
1  if (tf_buffer_.canTransform(
2          "base_footprint", "target",
3          tf2::TimePointZero, &error)) {
4
5          auto msg = tf_buffer_.
               lookupTransform(
6                "base_footprint", "target",
7                tf2::TimePointZero);
8
9          tf2::fromMsg(msg, bf2target);
10         double x = bf2target.getOrigin().x
               ();
11         double y = bf2target.getOrigin().y
               ();
12 }
```

# Transformar un punto entre frames (PointStamped + doTransform)

- Si calculas (x,y) en el frame del sensor, exprésalo como `PointStamped` y transfórmarlo a `base_link`.
- Usa el **stamp del sensor** para evitar incoherencias y extrapolaciones.
- El buffer TF compone e interpola; tú solo aplicas la transformación.

**ObstacleDetectorNode.cpp (real)**

```
geometry_msgs::msg::PointStamped p;
p.header = scan.header; // frame_id + stamp
p.point.x = obstacle_x;
p.point.y = obstacle_y;
p.point.z = 0.0;

try {
       auto tf = tf_buffer_.lookupTransform(
              "base_link", p.header.frame_id,
              p.header.stamp, tf2::durationFromSec(0.1)
                    );

       geometry_msgs::msg::PointStamped p_base;
       tf2::doTransform(p, p_base, tf);
} catch (const tf2::TransformException & ex) {
       RCLCPP_WARN(get_logger(), "TF failed: %s", ex.
             what());
}
```

# Imágenes en ROS 2: de `Image` a resultados (detección)

- Las imágenes llegan como `sensor_msgs/msg/Image`.
- En la mayoría de algoritmos, trabajas con `cv::Mat` (OpenCV).
- Patrón típico:
  1. `Image` → `cv::Mat` (cv_bridge)
  2. procesado (HSV, segmentación, etc.)
  3. publicar detección (`vision_msgs`)

# cv_bridge: Image → cv::Mat

- cv_bridge::toCvCopy convierte el mensaje ROS a cv::Mat.
- Se pide un encoding explícito (p.ej. BGR8).
- Puede fallar (encoding no convertible) → try/catch.

**HSVFilterNode.cpp (real)**

```
1 cv_bridge::CvImagePtr cv_ptr;
2 try {
3         cv_ptr = cv_bridge::toCvCopy(
4                 image, sensor_msgs::
                      image_encodings::BGR8);
5 } catch (cv_bridge::Exception & e) {
6         RCLCPP_ERROR(get_logger(), "... %s
                ", e.what());
7         return;
8 }
9 cv::Mat & image_cv = cv_ptr->image;
```

# Filtrado HSV: cvtColor + inRange

- En HSV, el color es más estable ante cambios de iluminación.
- inRange crea una máscara binaria (0/255) con los píxeles que cumplen el rango.
- Si la máscara está vacía: regla de la práctica **no publiques nada**.

**Procesado (real)**

```
1 cv::Mat img_hsv, out;
2 cv::cvtColor(in, img_hsv, cv::
      COLOR_BGR2HSV);
3 cv::inRange(img_hsv,
4         cv::Scalar(h,s,v), cv::Scalar(H,S,
              V), out);
```

# Visualización rápida: imshow + waitKey + copyTo

- Para depurar, puedes superponer la máscara sobre la imagen original.
- `waitKey(1)` procesa eventos GUI; sin ella `imshow` no refresca.
- En robot real (sin GUI) suele desactivarse o sustituirse por publicación en RViz.

**Patrón (real)**

```
1 cv::waitKey(1);
2
3 cv::Mat out_image;
4 image.copyTo(out_image, image_filtered);
5 cv::imshow("Filtered Image", out_image);
```

# Publicar una detección 2D (vision_msgs)

- Se rellena la cabecera (frame + timestamp) y el bounding box.
- Reutiliza `header.frame_id` y `header.stamp` de la imagen de entrada.
- Publica en un topic (p.ej. `/detection_2d`).

**Publicación (extracto real)**

```
1  vision_msgs::msg::Detection2D det;
2  det.header.frame_id = image->header.
       frame_id;
3  det.header.stamp = image->header.stamp;
4  det.bbox.center.position.x = ...;
5  det.bbox.center.position.y = ...;
6  det.bbox.size_x = bbx.width;
7  det.bbox.size_y = bbx.height;
8
9  vision_msgs::msg::Detection2DArray arr;
10 arr.header = det.header;
11 arr.detections.push_back(det);
12 detection_pub_->publish(arr);
```

# CameraInfo + pinhole: del píxel a un rayo 3D

- CameraInfo contiene intrínsecos (modelo pinhole).
- image_geometry::PinholeCameraModel encapsula rectificación y proyecciones.
- projectPixelTo3dRay devuelve una dirección 3D asociada a un píxel.

**Modelo + rayo (real)**

```
1  model_ = std::make_shared<
2         image_geometry::PinholeCameraModel
              >();
3  model_->fromCameraInfo(*info);
4
5  cv::Point3d ray = model_->
       projectPixelTo3dRay(
6        model_->rectifyPoint(center));
7  ray = ray / ray.z;
8  float yaw = atan2(ray.x, ray.z);
9  float pitch = atan2(ray.y, ray.z);
```

# Depth → 3D: idea y flujo de datos

- Si tienes imagen de profundidad, cada píxel aporta una profundidad $Z$.
- Flujo del ejemplo:
  `Detection2DArray` + `depth Image` + `CameraInfo` → `Detection3DArray`
- Problemas reales:
  - Mensajes llegan por topic(s) distintos y no están perfectamente sincronizados.
  - Depth puede venir en `16UC1` (mm) o `32FC1` (m) y con valores inválidos.

# message_filters: suscriptores sincronizables

- Para emparejar depth Image con Detection2DArray, no basta con callbacks separados.
- message_filters proporciona **suscriptores** que se conectan a un sincronizador.
- En el ejemplo se suscriben dos fuentes: input_depth y input_detection_2d.

**Suscriptores (real)**

```
1  depth_sub_ = std::make_shared<
2          message_filters::Subscriber<
3               sensor_msgs::msg::Image>>(
4          this, "input_depth",
           rclcpp::SensorDataQoS().reliable()
                .get_rmw_qos_profile());
5
6  detection_sub_ = std::make_shared<
7          message_filters::Subscriber<
               vision_msgs::msg::
               Detection2DArray>>(
8          this, "input_detection_2d",
9          rclcpp::SensorDataQoS().reliable()
                .get_rmw_qos_profile());
```

# message_filters: **ApproximateTime** + callback conjunto

- El sincronizador mantiene colas pequeñas y aplica una política **ApproximateTime**.
- Cuando encuentra dos mensajes con timestamps suficientemente cercanos, dispara un **callback conjunto**.
- Resultado: reduces emparejamientos erróneos (detección vs depth de otro instante).

**Sincronizador (real)**

```
1  sync_ = std::make_shared<
2    message_filters::Synchronizer<
         MySyncPolicy>>(
3    MySyncPolicy(10), *depth_sub_, *
         detection_sub_);
4
5  sync_->registerCallback(
6    std::bind(&DetectionTo3DfromDepthNode::
         callback_sync,
7              this, _1, _2));
```

# Leer profundidad: 16UC1 vs 32FC1

- `encoding` indica el tipo de los píxeles en el búfer.
- `16UC1`: `uint16_t` (típicamente mm) → convertir a metros.
- `32FC1`: `float` (típicamente m).
- Casos inválidos:
  - `16UC1`: a menudo 0 significa "sin retorno".
  - `32FC1`: pueden aparecer `NaN`/`Inf`.

# Lectura de $Z$ en el píxel $(u, v)$ (depth)

- El flujo mínimo es:
  1. elegir $(u, v)$ (centro de bbox)
  2. leer $Z$
  3. proyectar a (X,Y,Z)
- Comprueba límites de imagen ($0 \leq u <$ `width`, $0 \leq v <$ `height`).
- Si el valor no es válido, lo razonable es no publicar esa detección.

**Lectura (real)**

```
1  cv_bridge::CvImagePtr cv_depth_ptr =
2          cv_bridge::toCvCopy(*image_msg,
               image_msg->encoding);
3
4  float depth = 0.0;
5  if (image_msg->encoding == "16UC1") {
6          depth = depth_image_proc::
               DepthTraits<uint16_t>::
               toMeters(
7                 cv_depth_ptr->image.at<
                       uint16_t>(cv::Point2d(u
                       , v)));
8  } else {
9          depth = cv_depth_ptr->image.at<
               float>(cv::Point2d(u, v));
10 }
```

# Proyección pinhole: (u,v,depth) → (X,Y,Z)

- Fórmula clásica (intrínsecos $f_x, f_y, c_x, c_y$):

$$X = \frac{(u - c_x)Z}{f_x}, \quad Y = \frac{(v - c_y)Z}{f_y}$$

- El ejemplo usa `PinholeCameraModel`:
  - rayo 3D del píxel
  - normaliza para que $z = 1$
  - escala por `depth`

**Deproyección (real)**

```
1 cv::Point3d ray = model_->
      projectPixelTo3dRay(
2       model_->rectifyPoint(cv::Point2d(u
            , v)));
3 ray = ray / ray.z;
4 cv::Point3d point = ray * depth;
```

# PointCloud2 → 3D: cuando ya tienes (X,Y,Z)

- Alternativa: usar directamente `sensor_msgs/msg/PointCloud2`.
- Si la nube es **organizada** (como una imagen 3D), hay correspondencia píxel $(u, v) \rightarrow$ punto.
- Flujo: sincronizar nube + detección 2D → convertir a PCL → extraer punto central → publicar `Detection3DArray`.

# Convertir `PointCloud2` a PCL

- `PointCloud2` es un búfer binario.
- `pcl::fromROSMsg` lo convierte a `pcl::PointCloud<...>`.
- A partir de ahí puedes indexar/filtrar con APIs PCL.

**Conversión (real)**

```
1 pcl::PointCloud<pcl::PointXYZ>::Ptr pc(
2         new pcl::PointCloud<pcl::PointXYZ
              >);
3
4 pcl::fromROSMsg(*pc_msg, *pc);
```

# Extraer punto 3D del centro de la bbox (nube organizada)

- Si la nube es organizada, puedes indexar como matriz: `pc->at(u,v)`.
- Filtra puntos no finitos (`NaN/Inf`) antes de publicar.
- Si la nube **no** es organizada, este método no aplica directamente.

**Extracción (real)**

```
pcl::PointXYZ & center = pc->at(u, v);

detection_3d_msg.bbox.center.position.x =
    center.x;
detection_3d_msg.bbox.center.position.y =
    center.y;
detection_3d_msg.bbox.center.position.z =
    center.z;
```

## Resumen: dos formas de obtener 3D

- **Depth + CameraInfo:** lees $Z$ por píxel y deproyectas con pinhole (muy general).
- **PointCloud2:** ya tienes (X,Y,Z); si la nube es organizada, extraes el punto con `at(u,v)`.
- En ambos casos: si no hay dato válido, **no publiques** (regla coherente con la práctica).

# Preguntas